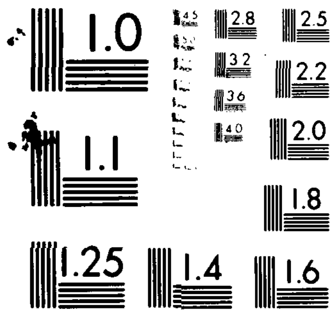
AD-A108 585 MARYLAND UNIV COLLEGE PARK DEPT OF COMPUTER SCIENCE F/G 12/1
GENERALIZING SPECIFICATIONS FOR UNIFORMLY IMPLEMENTED LOOPS.(U)
OCT 81 D D DUNLOP, V R BASILI F49620-80-C-0001
UNCLASSIFIED TR-1116 AFOSR-TR-81-0791 NL

END
DATE
FILMED
1 82
DTIC



MICROCOPY RESOLUTION TEST CHART

NATIONAL BUREAU OF STANDARDS 1963-A

AFOSR-TR- 81-0791

LEVEL II

(12)

AD A108585

COMPUTER SCIENCE TECHNICAL REPORT SERIES

DTIC ELECTE DEC 15 1981 S D H

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

FILE COPY

Approved for public release; distribution unlimited.

81 12 14 036

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFOSR-TR- 81 -0791	2. GOVT ACCESSION NO. ADA108585	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) GENERALIZING SPECIFICATIONS FOR UNIFORMLY IMPLEMENTED LOOPS		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER TR-1116
7. AUTHOR(s) Douglas D. Dunlop and Victor R. Basili		8. CONTRACT OR GRANT NUMBER(s) F49620-80-C-0001
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Computer Science University of Maryland College Park, Maryland 20742		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61102F; 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS Mathematical & Information Sciences Directorate Air Force Office of Scientific Research Bolling AFB DC 20332		12. REPORT DATE October 1981
		13. NUMBER OF PAGES 37
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

program verification, valid generalization, base generalization, uniformly implemented loop, iteration condition

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The problem of generalizing functional specifications for WHILE loops is considered. This problem occurs frequently when trying to verify that an initialized loop satisfies some functional specification, i.e. produces outputs which are some function of the program inputs.
The notion of a valid generalization of a loop specification is defined. A particularly simple valid generalization, a base generalization, is discussed. A property of many commonly occurring WHILE loops, that of being uniformly implemented, is defined. A technique is presented which exploits this property

in order to systematically achieve a valid generalization of the loop specification. Two classes of uniformly implemented loops which are particularly susceptible to this form of analysis are defined and discussed. The use of the proposed technique is illustrated with a number of applications. Finally, an implication of the concept of uniform loop implementation for the validation of the obtained generalization is explained.

A

Technical Report TR-1116 October, 1981
[redacted]-F49620-80-C-[redacted]
0001

Generalizing Specifications For Uniformly Implemented Loops*

Douglas D. Dunlop and Victor R. Basili

Department of Computer Science
University of Maryland
College Park, MD 20742

DTIC
SELECTE
DEC 1 5 1981
S D
H

AIR FORCE OFFICE OF SCIENTIFIC RESEARCH (AFSC)
NOTICE OF TRANSMITTAL TO DTIC
This technical report has been reviewed and is approved for public release IAW AFR 190-12.
Distribution is unlimited.
MATTHEW J. KERPER
Chief, Technical Information Division

---

*This work was supported in part by the Air Force Office of Scientific Research Contract [redacted]-F49620-80-C-0001 to the University of Maryland. The material contained in this paper will become part of a dissertation to be submitted to the Graduate School, University of Maryland, by Douglas D. Dunlop, in partial fulfillment of the requirements for the Ph.D. degree in Computer Science.

© Copyright 1981 by D.D. Dunlop and V.R. Basili

4

ABSTRACT

The problem of generalizing functional specifications for WHILE loops is considered. This problem occurs frequently when trying to verify that an initialized loop satisfies some functional specification, i.e. produces outputs which are some function of the program inputs.

The notion of a valid generalization of a loop specification is defined. A particularly simple valid generalization, a base generalization, is discussed. A property of many commonly occurring WHILE loops, that of being uniformly implemented, is defined. A technique is presented which exploits this property in order to systematically achieve a valid generalization of the loop specification. Two classes of uniformly implemented loops which are particularly susceptible to this form of analysis are defined and discussed. The use of the proposed technique is illustrated with a number of applications. Finally, an implication of the concept of uniform loop implementation for the validation of the obtained generalization is explained.

KEYWORDS and PHRASES: program verification, valid generalization, base generalization, uniformly implemented loop, iteration condition

CR CATEGORIES: 5.24

Generalizing Specifications For Uniformly Implemented Loops

1. Introduction

Consider the problem of proving/disproving a WHILE loop correct with respect to some functional specification f, i.e. f requires the output variable(s) to be some function of the inputs to the loop. If the loop precondition is weak enough so that the domain of f contains the intermediate states which appear after each loop iteration, the loop is said to be closed for the domain of f. An important result in program verification is that if the loop is closed for the domain of its specification, there are two easily constructed verification conditions based solely on the specification, loop predicate and loop body which are necessary and sufficient conditions for the partial correctness of the loop with respect to its specification [Mills 75, Misra 78]. If the loop is not closed for the domain of the specification function, a generalized specification (i.e. one that implies the original specification) which satisfies the closure requirement must be discovered before these verification conditions can be constructed (this problem is analogous to that of discovering an adequate loop invariant for an inductive assertion proof [Hoare 69] of the program).

We remark that the restricted specification often occurs in the process of analyzing an initialized WHILE loop, i.e. one that consists of a WHILE loop preceded by some initialization code. This initialization typically takes the form of assignments of constant values to some of the variables manipulated by the loop. Examples include setting a counter to zero, a search flag to

FALSE, a queue variable to some particular configuration, etc. It is clear that the initialized loop is correct with respect to some specification if and only if the WHILE loop by itself is correct with respect to a slightly modified specification. This specification has the same postcondition as the original specification and a precondition which is the original precondition together with the condition that the initialized variables have their initialized values. Since the initialized variables will typically assume other values as the loop iterates, the loop most likely will not be closed for the domain of this specification and a generalization of it will be necessary in order to verify the correctness of the program.

Example 1 - The following program multiplies natural numbers using repeated addition:

```
{v>=0,k>=0}
z := 0;
while v > 0 do
   z := z + k;
   v := v - 1
   od
{z=v0*k}.
```

Accession For
NTIS GRA&I ☑
DTIC TAB ☐
Unannounced ☐
Justification
By
Distribution/
Availability Codes
Avail and/or
Dist Special
A

The term v0 appearing in the postcondition refers to the initial value of v. The program is correct if and only if

```
{z=0,v>=0,k>=0}
while v > 0 do
   z := z + k;
   v := v - 1
   od
{z=v0*k}
```

is correct. Since this loop precondition requires z to have the value 0 and z assumes other values as the loop executes, the loop is not closed for this precondition. Thus, before this program can be verified using the above mentioned technique, this specification must be generalized to something like

```
{v>=0,k>=0}
while v > 0 do
   z := z + k;
   v := v - 1
   od
{z=z0 + v0*k}
```

where z0 refers to the initial value of the variable z.

The approach to this problem suggested here is one of observing how particular changes in the value of some input variable (e.g. z in the example) affect the result produced by the loop body of the loop under consideration. Clearly in general, a change in the value of an input variable may cause an arbitrary (and seemingly unrelated) change in the loop body result. In many commonly occurring cases, however, the result produced by the loop body is "uniform" across the entire spectrum of possible values for the input variable. It is this property that will be exploited in order to obtain a generalized specification for the loop being analyzed. The generalizations considered here have the property that the loop is correct with respect to the generalization if and only if the loop is correct with respect to the original specification. Thus if the loop is closed for the domain of the generalization, the program can be proven/disproven by testing its correctness relative to the generalization.

Generalizing Specifications For Uniformly Implemented Loops

We remark that the general problem of finding a suitable generalized loop specification has been shown to be NP-complete [Wegbreit 77], i.e. it appears quite unlikely that there will ever exist an easily applied procedure for obtaining such generalizations that will work in all cases. On the other hand, work presented here and elsewhere [Basu & Misra 76, Misra 79, Basu 80], indicates that generalized specifications can be obtained in a systematic manner for restricted classes of loops. We feel that the notion of "uniform" loop body behavior discussed in this paper is valuable not only as a tool by which such generalizations may be obtained, but also as an attempt at a characterization of loops which are susceptible to routine analysis, and hence in this sense, easy to verify and comprehend.

The following section defines the necessary notation and terminology and then introduces the idea of a generalized loop specification. Section 3 defines a uniformly implemented loop and states several implications of this definition for the problem of generalizing a specification for such a loop. These results are applied on several example programs in Section 4. In Section 5, a simplified procedure is suggested for proving/disproving a uniformly implemented loop correct with respect to the obtained generalization. Finally, several guidelines for recognizing uniformly implemented loops are presented in Section 6.

2. Preliminaries

We will consider a verification problem of the form

```
{<z,X> ∈ D(f)}
while B(z,X) do
    z,X := h´(z,X),h´´(z,X)
    od
{<z,X> = f(<z0,X0>)}.
```

In this problem, f is a data state to data state function. The data state consists of two variables, z and X. The notation D(f) means the set of states in the domain of f (i.e. the set of states for which f is defined). The terms z0 and X0 refer to the initial values of z and X respectively. The effect of the loop body is partitioned into two functions h´ and h´´ which describe the new values of z and X respectively.

The loop will be referred to as P. The data state to data state function computed by the loop (which, presumably, is not explicitly known) will be denoted [P]. Thus D([P]) is the set of states for which P terminates. As a shorthand notation we will use Y for the state <z,X>, and H for the data state to data state function computed by the loop body, i.e.

$$H(Y) = H(<z,X>) = <h'(z,X),h''(z,X)>.$$

Suppose the loop is not closed for D(f) in that this set contains only a restricted collection of values (maybe only one) of z and that other intermediate values of z occur as the loop iterates. The variable z will be called the key variable. Our goal here is to discover some more general specification f´ which includes each of these intermediate values of the key variable in its domain. This generalization process (in one form or another) is necessary for a proof of correctness of the program under consideration.

Definition - P is correct with respect to (wrt) a function f if and only if (iff) for all Y in D(f), [P](Y) is defined and [P](Y)=f(Y).

Definition - A superset f′ of f is a valid generalization of f iff if P is correct wrt f, then P is correct wrt f′.

Note that the collection of supersets of f is partially ordered by "is a valid generalization of." The following theorem defines one technique for constructing a valid generalization of the specification function f.

Theorem 1 - A superset g of f whose extension is defined by

(1) $\tilde{}B(Y) \rightarrow g(Y)=Y$

is a valid generalization of f.

Proof - Suppose P is correct wrt f. Let Y ∈ D(g). If Y ∈ D(f), the loop handles the input correctly by hypothesis. If Y is not in D(f), we must have ~B(Y) and g(Y)=Y. Thus the program and g map Y to itself and thus are in agreement. Consequently P is correct wrt g, and g is a valid generalization of f.

The theorem utilizes the fact that the loop must necessarily compute the identity function over inputs where the loop predicate is false. Combining this information with the program specification f results in a valid generalization of f.

We note that if there does not exist a superset g of f whose extension is defined by (1), the theorem is vacuously true. This would occur if an element Y from the domain of f satisfied $\sim B(Y)$ as well as $f(Y) \neq Y$ (this would imply that the program was not correct wrt f). If there does exist a superset g of f whose extension is defined by (1), the superset is unique. Throughout this report, we will refer to the function g as the base generalization of the specification f.

Definition - A valid generalization f′ of f is adequate if the loop is closed for D(f′).

The important characteristic of an adequate valid generalization f′ is that it can be used to prove/disprove the correctness of P wrt the original specification f. Since the loop is closed for D(f′), P can be proven/disproven correct wrt f′ using standard techniques [Mills 72, Mills 75, Basu & Misra 75, Morris & Wegbreit 77, Wegbreit 77, Misra 78]. Specifically, P is correct wrt f′ iff each of

(2) the loop terminates for all $Y \in D(f')$

(3) $Y \in D(f')$ & $\sim B(Y) \rightarrow f'(Y)=Y$

(4) $Y \in D(f')$ & $B(Y) \rightarrow f'(Y)=f'(H(Y))$

hold. If P is correct wrt f′, then P is necessarily correct wrt any subset of f′, including f. If P is not correct wrt f′, then by the definition of a valid generalization, P must not be correct wrt f.

Example 2 - The following program tests whether a particular key appears in an ordered binary tree.

```
{success=FALSE}
while tree ≠ NULL and ~success do
   if     name(tree) = key then success := TRUE
   elseif name(tree) < key then tree := right(tree)
   else                         tree := left(tree) fi
   od
{success = IN(tree0,key)}
```

The function IN(tree0,key) appearing in the postcondition is a predicate which means "the ordered binary tree tree0 contains a node with name field key." The boolean variable success is chosen as the key variable since it is constrained to the value FALSE in the input specification. Thus success plays the role of z and the pair of variables <tree,key> correspond to X in the program schema discussed above. The specification function f is

f(<FALSE,tree,key>) = <IN(tree,key),tree´,key´>

where tree´ and key´ are the final values of the variables tree and key computed by the loop, respectively. That is, since the final values of these variables are not of interest in this example, we specify these final values so as to be automatically correct. Using Theorem 1, a valid generalization of this specification is

```
g(<success,tree,key>) = if ~success then
                             <IN(tree,key),tree´,key´>
                        else if tree=NULL or success then
                             <success,tree,key>,
```

which is equivalent to

g(<success,tree,key>) = <success or IN(tree,key),tree´,key´>.

In this example, the domain of the base generalization g of f includes each value of the key variable, (i.e. FALSE and TRUE) and is thus adequate. Consequently, this generalization can be used to prove/disprove the correctness of the program.

In most cases, however, the heuristic suggested in the theorem is insufficient to generate an adequate generalization. Indeed, the base generalization is an adequate generalization only in the case when the sole reason for the closure condition not holding is the existence of potential final values of the key variable (e.g. TRUE in the example) which are absent from D(f). In order to obtain a generalization that includes general values of the key variable, an important characteristic of the loop body which seems to be present in many commonly occurring loops will be exploited.

3. Uniformly Implemented Loops

Definition - Let P be a loop of the form described above. Let A be a set, and let Z be the set of values the key variable z may assume. Let

$$\$' : A \times Z \rightarrow Z$$

be an infix binary operator. The loop P is uniformly implemented with respect to (wrt) $\$'$ iff each of

(5) $B(z,X) \rightarrow h'(a \ \$' \ z,X) = a \ \$' \ h'(z,X)$

(6) $B(z,X) \rightarrow h''(a \ \$' \ z,X) = h''(z,X)$

(7) $B(z,X) \rightarrow B(a \ \$' \ z,X)$

hold.

Conditions (5) and (6) of this definition state that a modification to the key variable by the operation $\$'$ causes a <u>slight</u> but <u>orderly</u> change in the result produced by the loop body. The change is slight because the only difference in the result produced by the loop body occurs in the key variable. This difference is orderly because it corresponds precisely to the same $\$'$ operation that served to modify the input value of the key variable. Condition (7) specifies that such a modification does not cause the loop predicate B to change from TRUE to FALSE.

As a shorthand notation we define the infix operator \$ as

$$a \$ Y = a \$ <z,X> = <a \$' z,X>.$$

In this notation (5)-(7) are equivalent to

(8) $B(Y) \rightarrow a \$ H(Y) = H(a \$ Y)$

and

$$B(Y) \rightarrow B(a \$ Y).$$

<u>Example 3</u> - Consider again the program from Example 1 which multiplies natural numbers using repeated addition:

```
{z=0,v>=0,k>=0}
while v>0 do
    z := z + k;
    v := v - 1
    od
{z=v0*k}.
```

Let z be the key variable. The pair <v,k> corresponds to the variable X occurring in the above schema. The loop is uniformly implemented wrt +, where A and Z are both the set of natural numbers. Note that adding some constant to the input value of z has the effect of adding the same constant to the value of z output by the loop body. Now consider the following alternative implementation of multiplication:

```
{z=0,v>=0,k>=0}
while v>0 do
    if z<k then        z := z + k
    elseif z=k then    z := z * 2 * v
    else               z := z - k fi;
    v := v - 1
    od
{z=$\overline{v0}$*k}.
```

Again, let z be the key variable. This loop is not uniformly implemented wrt +. Intuitively, this is due to the high degree of dependence of the loop body behavior on the value of the key variable. The result of this dependence is that adding some constant to the value of z causes an unorderly change in the value of z output by the loop body.

The reader may wonder if the second multiplication program above might be uniformly implemented wrt some operation other than +. We remark that <u>any</u> loop is uniformly implemented wrt $' : A x Z -> Z defined by

a $' z = z

for all a ∈ A and z ∈ Z. For the purpose of this report, we rule out such trivial operations, i.e. we require that for any z ∈ Z, there exists some a ∈ A such that

a $' z ≠ z.

With this assumption, there does not exist an operation wrt which the second of the above loops is uniformly implemented (or more briefly, the loop is not uniformly implemented). To see this, suppose the loop were uniformly implemented wrt $\$' : A \times Z \to Z$. Select $z=0$, $a \in A$ such that $a \$' 0 \neq 0$, and $k = a \$' 0$. We evaluate condition (5) as follows:

$$\begin{array}{llll}
 & B(z,X) & \to h'(a \$' z,X) & = a \$' h'(z,X) \\
\text{i.e.} & v > 0 & \to h'(a \$' 0,\langle v,k\rangle) & = a \$' h'(0,\langle v,k\rangle) \\
\text{i.e.} & v > 0 & \to h'(k,\langle v,k\rangle) & = a \$' h'(0,\langle v,k\rangle) \\
\text{i.e.} & v > 0 & \to k*2*v & = a \$' h'(0,\langle v,k\rangle)
\end{array}$$

which implies

$$v>0 \;\&\; k>0 \to k*2*v \qquad = a \$' k.$$

Since the term $k*2*v$ will vary with different values of v where k is positive, and $a \$' k$ is independent of v, this condition is false and thus (5) does not hold. We conclude that the second multiplication program above is not uniformly implemented. That is, there does not exist a nontrivial modification that can be applied to the variable z which always results in a slight and orderly change in the result produced by the loop body.

The results presented here are based on the following lemma concerning uniformly implemented loops. The lemma describes the output of the loop for some modified input $a \$ Y$ (i.e. $[P](a \$ Y)$) in terms of the output of the loop for the input Y (i.e. $[P](Y)$) and the output of the loop for the input $a \$ [P](Y)$ (i.e. $[P](a \$ [P](Y))$).

Lemma 1 - Let P be uniformly implemented wrt $\$'$. Then

(9) $Y \in D([P]) \to [P](a \$ Y) = [P](a \$ [P](Y))$.

Proof - We use induction on the number of iterations of P on Y. For the base case of 0 iterations, [P](Y)=Y, and the lemma holds. Suppose it holds for Y values requiring n-1 iterations where n > 0. Let Y1 require n iterations. Since n > 0, B(Y1) holds. By (7), B(a $ Y1). Note that H(Y1) requires n-1 iterations on P; thus by the inductive hypothesis

[P](a $ H(Y1)) = [P](a $ [P](H(Y1))).

Due to the uniform implementation this is

[P](H(a $ Y1)) = [P](a $ [P](H(Y1))).

Using the loop property B(Y) -> [P](Y)=[P](H(Y)) on both sides we get

[P](a $ Y1) = [P](a $ [P](Y1)).

Thus the inductive step holds and the lemma is proved.

The general idea behind our use of the lemma is as follows. Suppose the value [P](Y) is known for some particular Y. I.e. suppose we know what the loop produces for the input Y. In addition, suppose that, given the result [P](Y), the quantity [P](a $ [P](Y)) is also known. With this information, we can then use Lemma 1 to "solve" for the (possibly unknown) value [P](a $ Y). This additional information concerning the input/output behavior of the loop can be used as an aid in constructing a valid generalization of the specification f.

How can we find the value [P](Y) and then the value [P](a $ [P](Y)) for some Y? The key lies in assuming the loop P is correct wrt f. If P is not correct wrt f, any generalization of f obtained by the technique will be a valid generalization by definition. Under this assumption, $[P](Y)$ is known for $Y \in D(f)$, i.e. $Y \in D(f) \rightarrow [P](Y)=f(Y)$, and hence Lemma 1 implies

(10) $Y \in D(f) \rightarrow [P](a \$ Y)=[P](a \$ f(Y))$.

Consider now the base generalization g of f defined in Theorem 1. Recall that g is simply f augmented with the identity function over the domain where the loop predicate B is false. Assuming as before that P is correct wrt f, P is then correct wrt g by Theorem 1; hence $Y \in D(g) \rightarrow [P](Y)=g(Y)$. Thus (10) implies

(11) $Y \in D(f) \ \& \ a \$ f(Y) \in D(g) \rightarrow [P](a \$ Y)=g(a \$ f(Y))$.

Thus we can "solve" for the behavior of the loop on the input a \$ Y, assuming $Y \in D(f)$, $a \$ f(Y) \in D(g)$ and P is correct wrt f. This suggests that the superset f' of f whose extension is defined by

(12) $Y \in D(f) \ \& \ a \$ f(Y) \in D(g) \rightarrow f'(a \$ Y)=g(a \$ f(Y))$

is a valid generalization of f. Before giving a formal proof of this result, however, we first consider the question of the existence of such a superset. Specifically, it could be that for some a and Y satisfying $Y \in D(f)$ and $a \$ f(Y) \in D(g)$, that $a \$ Y \in D(f)$ and $f(a \$ Y) \neq g(a \$ f(Y))$, which would imply $f(a \$ Y) \neq f'(a \$ Y)$. In this case, a valid generalization of f based on (12) cannot exist (it would have to be ambiguously defined). The following theorem states that this implies P is not correct wrt f.

<u>Theorem 2</u> - If P is correct wrt f, there exists a superset f' of f whose extension is defined by (12), i.e.

$Y \in D(f) \ \& \ a \$ f(Y) \in D(g) \rightarrow f'(a \$ Y)=g(a \$ f(Y))$.

Proof - Let f' be the function computed by the loop, i.e. $[P]$. Since P is correct wrt f, P is correct wrt g, and f' is a superset of both f and g. By the lemma

$$f'(a \$ Y) = f'(a \$ f'(Y))$$

for all $Y \in D(f)$. Since $f'(Y)=f(Y)$ for $Y \in D(f)$ and $f'(Y)=g(Y)$ for $Y \in D(g)$, (12) holds. The subset of f' which contains f and whose extension is defined by (12) satisfies the theorem.

The following theorem is the central result presented here. The theorem formalizes the use of Lemma 1 in the manner suggested above, i.e. that the superset described in the previous theorem is a valid generalization of the original specification.

Theorem 3 - A superset f' of f whose extension is defined by (12), i.e.

$$Y \in D(f) \ \& \ a \$ f(Y) \in D(g) \rightarrow f'(a \$ Y)=g(a \$ f(Y)),$$

is a valid generalization of f.

Proof - Suppose P is correct wrt f. Let $Y \in D(f)$ and $a \$ f(Y) \in D(g)$. By Lemma 1 $[P](a \$ Y) = [P](a \$ [P](Y))$. Since P is correct wrt f this is $[P](a \$ Y) = [P](a \$ f(Y))$. By Theorem 1, P is correct wrt g. Using this, the equality can be written as $[P](a \$ Y) = g(a \$ f(Y))$. Substituting using (12) yields $[P](a \$ Y) = f'(a \$ Y)$. Thus P and f' are in agreement on the input $a \$ Y$ and consequently are in agreement on any input in $D(f')$. Hence P is correct wrt f' and thus f' is a valid generalization of f.

Generalizing Specifications For Uniformly Implemented Loops

The significance of Theorem 3 is that it provides a guideline for generalizing the specification of a uniformly implemented loop. If the loop is closed for the domain of the resulting specification, the generalization can then be used to prove/disprove the program correct wrt the original specification.

4. Applications

In this section we illustrate the use of Theorem 3 with a number of example programs which fall into either of two subclasses of uniformly implemented loops. The subclasses correspond to the two possible circumstances which can occur when a \$ f(Y) of condition (12) belongs to the set D(g): the first, because ~B(a \$ f(Y)), and, the second, because a \$ f(Y) $\in$ D(f). In each of these situations, condition (12) takes on a particularly simple form.

Definition - A uniformly implemented loop satisfying

~B(Y) -> ~B(a \$ Y)

is a Type A loop.

Observe that this condition along with (7) indicates that a Type A uniformly implemented loop satisfies

B(Y) <-> B(a \$ Y),

i.e. the value of the loop predicate B is independent of a change to the data state by the operator \$.

Generalizing Specifications For Uniformly Implemented Loops

The intuition behind a Type A uniformly implemented loop is as follows. Whenever an execution of a Type A loop terminates (i.e. $\neg B(Y)$ holds) and the resulting data state is modified by the operator \$, the result is a new data state which, when viewed as a loop input, corresponds to zero iterations of the loop (i.e. the predicate B is still FALSE despite the modification). This property is reflected in the following corollary.

Corollary 1 - Let P be a Type A loop. A superset f' of f whose extension is defined by

(13) $Y \in D(f) \rightarrow f'(a \$ Y)=a \$ f(Y)$

is a valid generalization of f.

Proof - The proof consists of showing that (12) and (13) are equivalent for a Type A loop which is correct wrt f. By Theorem 3, the corollary then holds. Let P be a Type A loop which is correct wrt f. A consequence of the correctness property is that $\neg B(f(Y))$ for all $Y \in D(f)$. Since P is a Type A loop, this implies $\neg B(a \$ f(Y))$. Thus $a \$ f(Y) \in D(g)$ and $g(a \$ f(Y))=a \$ f(Y)$. Consequently (12) and (13) are equivalent.

Of course, once a generalization f' has been obtained via Corollary 1, there is no reason why that result cannot be fed back into the corollary to obtain a (possibly) further generalization f'' (using f' for f, f'' for f'). This notion suggests the following general case of Corollary 1.

Corollary 2 - Let P be a Type A loop. A superset f^* of f whose extension is defined by

$Y \in D(f)$ & $n>0$ ->

f´(a1\$(a2\$ ... (an\$Y) ...))=a1\$(a2\$ ... (an\$f(Y)) ...)

is a valid generalization of f.

<u>Example 4</u> - Consider the following program to compute exponentiation.

```
{w=1,c>0,d>=0}
while d > 0 do
    if odd(d) then w := w * c fi;
    c := c*c; d:=d/2
    od
{w=c0 ^ d0}
```

The infix operator ^ appearing in the postcondition represents integer exponentiation. In this example, w plays the role of the key variable z, and the pair <c,d> corresponds to the variable X. We now consider wrt what operation the loop might be uniformly implemented. For any operation \$´, (7) holds (because w does not appear in the loop predicate) as does (6) (because the values produced in c and d are independent of w). Furthermore, (5) must hold for inputs which bypass the updating of w. Thus the uniformity conditions reduce to

d > 0 & odd(d) -> (a \$´ w) * c = a \$´ (w * c)

Due to its associativity, it is clear the loop is uniformly implemented wrt *, where the sets A and Z are the set of integers. Since the key variable does not appear in the loop predicate, it is necessarily a Type A loop. Let c>0 and d>=0. The specification function here is

f(<1,c,d>) = <c ^ d,c´,d´>

where c´ and d´ are the final values computed by the loop for the variables c and d. Applying Corollary 1,

$$f'(<a*1,c,d>) = <a*(c \hat{} d),c',d'>$$

is a valid generalization of f. Since this holds for all a, the definition of f′ can be rewritten as

$$f'(<w,c,d>) = <w*(c \hat{} d),c',d'>.$$

The generalization f′ is adequate and can thus be used to test the correctness of the program wrt the original specification. Applying (2), (3) and (4) from above, these necessary and sufficient verification conditions are

- the loop terminates for all c>0, d>=0,
- d=0 -> w=w*(c ^ d), and
- w*(c ^ d) is a loop constant (i.e. c0 ^ d0 = w*(c ^ d) is a loop invariant),

respectively. In Section 5, we will discuss a simplification of the last of these verification conditions which applies for uniformly implemented loops.

<u>Example 5</u> [Misra 79] - The following program constructs the preorder traversal of a binary tree with root node r. The program uses a stack variable st and records the traversal in a sequence variable seq.

```
{seq=NULL, st=(r) /* stack st contains only the root node r */}
while st ≠ EMPTY do
    p <= st; /* pop the top off the stack */
    seq := seq || name(p); /* concatenate name of p to seq */
    if right(p) ≠ NIL then st <= right(p) fi; /* push onto st */
    if left(p) ≠ NIL then st <= left(p) fi
    od
{seq=PREORDER(r)}
```

The function PREORDER(r) appearing in the postcondition is the sequence consisting of the preorder traversal of the binary tree with root node r. Let seq be the key variable. The same reasoning employed in the previous example indicates here that the loop is uniformly implemented wrt ||, where the sets A and Z are the set of all strings. It is a Type A loop. The specification function is

f(<NULL,(r)>) = <PREORDER(r),st´>.

Again, the ´ notation is used to represent the final values of variables that are of no interest. Applying Corollary 1 we obtain

f´(<seq,(r)>) = <seq||PREORDER(r),st´>

as a valid generalization of f. In this case, f´ is not adequate since it does not specify a behavior of the loop for arbitrary values of the stack st. We will return to this example after considering another subclass of uniformly implemented loops.

<u>Definition</u> - A uniformly implemented loop satisfying

$\sim B(Y) \rightarrow a \$ Y \in D(f)$

is a <u>Type B</u> loop.

The intuition behind a Type B uniformly implemented loop is as follows. Whenever an execution of a Type B loop terminates (i.e. ~B(Y) holds) and the resulting data state is modified by the operator $, the result is a new data state which is a "valid" starting point for a new execution of the loop (i.e. this new state is in D(f)). This property is reflected in the following corollary.

<u>Corollary 3</u> - Let P be a Type B loop. A superset f' of f whose extension is defined by

$$(14) \quad Y \in D(f) \rightarrow f'(a \$ Y) = f(a \$ f(Y))$$

is a valid generalization of f.

<u>Proof</u> - The proof consists of showing that (12) and (14) are equivalent for a Type B loop which is correct wrt f. By Theorem 3, the corollary then holds. Let P be a Type B loop which is correct wrt f. A consequence of the correctness property is that $\sim B(f(Y))$ for all $Y \in D(f)$. Since P is a Type B loop, this implies $a \$ f(Y) \in D(f)$. Thus $a \$ f(Y) \in D(g)$ and $g(a \$ f(Y)) = f(a \$ f(Y))$. Consequently (12) and (14) are equivalent.

As before, a general case of this corollary can be stated which corresponds to an arbitrary number of its applications.

<u>Corollary 4</u> - Let P be a Type B loop. A superset f' of f whose extension is defined by

$$Y \in D(f) \ \& \ n>0 \rightarrow$$
$$f'(a1\$(a2\$(\ldots\$(an\$Y)\ldots))) = f(a1\$f(a2\$f(\ldots\$f(an\$f(Y))\ldots)))$$

is a valid generalization of f.

<u>Example 5</u> (<u>continued</u>) - We now consider the problem of further generalizing the derived specification in the previous example. The variable for which the loop is not closed, st, will now be the key variable. Consider an operation $a \$' st$ that has the effect of adding an element a to the stack st. Before being more precise about this operation, we consider how the loop body works, and how its output depends on the value of the key vari-

able st.

We observe that the loop body behavior relies heavily on the characteristics of the node on the top of the stack. Consequently, a modification a $' st to st which pushed a new node a onto the top of st would not cause a slight and orderly change in the result produced by the loop body and the uniformity conditions (5)-(7) would not hold. However, because the loop body behavior seems to be independent of what lies underneath the top of the stack, we suspect the loop is uniformly implemented wrt ADDUNDER, where A is the set of binary tree nodes, Z is the set of stacks of binary tree nodes, and a ADDUNDER st is the stack that results from adding a to the bottom of st. Conditions (5)-(7) for this operation indicate that, indeed, this is the case.

Let f be the generalization f' from the previous example. In keeping with the convention described above, since st is now the key variable, we will reverse the order in which the two variables appear in the data state, i.e. we will write <st,seq> instead of <seq,st>.

The program is a Type B uniformly implemented loop since

st=EMPTY -> <a ADDUNDER st,seq> ∈ D(f)

where a is a node of a binary tree, and specifically

(15) st=EMPTY -> f(<a ADDUNDER st,seq>)=<st',seq||PREORDER(a)>.

Applying Corollary 4, if (r,an, ... ,al) is an arbitrary stack (with r on top, al on the bottom)

f'(<(r,an, ... ,al),seq>) =

f´(a1\$(a2\$(... \$(an\$<(r),seq>) ...))) =

f(a1\$f(a2\$f(... \$f(an\$f(<(r),seq>)) ...))) =

f(a1\$f(a2\$f(... \$f(an\$<st´,seq||PREORDER(r)>) ...))).

Recall that st´ refers to the final value of st computed by the loop. The loop predicate indicates this will always be the value EMPTY. Hence (15) can be applied from inside out giving

f(a1\$f(a2\$f(... \$<st´,seq||PREORDER(r)||PREORDER(an)> ...)))

= ... =

<st´,seq||PREORDER(r)||PREORDER(an)|| ... ||PREORDER(a1)>.

This resulting specification can be used to prove the correctness of the program.

Example 6 [Gries 79] - The following program computes Ackermann´s function using a sequence variable s of natural numbers. The notation s(1) is the rightmost element of s and s(2) is the second rightmost, etc. The sequence s(..3) is s with s(2) and s(1) removed.

```
{s=<m,n>,m>=0,n>=0}
while size(s) ≠ 1 do
    if     s(2)=0 then s:=s(..3)||<s(1)+1>
    elseif s(1)=0 then s:=s(..3)||<s(2)-1,1>
    else              s:=s(..3)||<s(2)-1,s(2),s(1)-1> fi
    od
{s=<A(m,n)>}
```

The function A(m,n) appearing in the postcondition in Ackermann´s function. The specification function is

f(<s(2),s(1)>)=<A(s(2),s(1))>.

Let s be the key variable. As the loop body behavior is independent of the leftmost portion of s, the loop is uniformly implemented wrt |, where A is the set of natural numbers, Z is the set of sequences of natural numbers, and a|s = <a>||s. The program is also a Type B loop. By Corollary 4,

f´(<s(n),s(n-1), ... ,s(1)>) =

f´(s(n)\$(s(n-1)\$(... \$(s(3)\$<s(2),s(1)>) ...))) =

f(s(n)\$f(s(n-1)\$f(... \$f(s(3)\$f(<s(2),s(1)>)) ...))) =

f(s(n)\$f(s(n-1)\$f(... \$f(s(3)\$<A(s(2),s(1)>) ...))) =

f(s(n)\$f(s(n-1)\$f(... \$f(<s(3),A(s(2),s(1))>) ...))) =

f(s(n)\$f(s(n-1)\$f(... \$<A(s(3),A(s(2),s(1)))> ...)))

= ... =

<A(s(n),A(s(n-1), ... ,A(s(3),A(s(2),s(1))) ...))>

is a valid generalization of f.

5. Simplifying the ´Iteration Condition´

The view of WHILE loop verification presented here is one of a two step process, the first step being the discovery of an adequate valid generalization f´ of the loop specification f, the second being the proof of 3 basic conditions (i.e. (2)-(4)) based on this generalization. We have seen that the uniform nature of a loop implementation may be used in the first step as an aid in discovering an appropriate generalization. In this section, we will exploit the same loop characteristic to substantially simplify one of the conditions which must be proven in the second step of this process.

The verification condition of interest is (4) above, i.e.

Y ∈ D(f´) & B(Y) -> f´(Y)=f´(H(Y)),

and is labeled the iteration condition in [Misra 78]. This condition assures that as the loop executes, the intermediate values of Y remain in the same level set of f′, i.e. the value of f′ is constant across the loop iterations. Previously we argued that if P is uniformly implemented wrt $′, a change in the key variable by $′ causes a slight but orderly change in the result produced by H. Roughly speaking then, the behavior of H is largely independent of the key variable. If f′ is chosen so as to be equally independent of the key variable, and the above condition holds for Y=<z,X> where X is arbitrary but the key variable z has a specific simple value, we might expect the condition to hold for all Y. Such an expectation would be based on the belief that the truth or falsity of this condition would also be largely independent of the key variable.

We formally characterize this circumstance in the following definition.

Definition - Let P be a loop of the form described above. A generalization f′ of f is represented by f iff

(16) $Y \in D(f\ ) \ \&\ B(Y) \rightarrow f\ (Y)=f'(H(Y))$

$\rightarrow$

(17) $Y \in D(f') \ \&\ B(Y) \rightarrow f'(Y)=f'(H(Y))$.

Thus if f′ is represented by f, condition (16) can be used in place of the iteration condition (17) in proving the loop is correct wrt f′ (and hence wrt f). The significance of this situation is that the iteration condition can be tested with the key variable constrained by initialization (as prescribed in $D(f)$). In practice, the result is one of having to prove a substantially simpler verification condition.

The following theorems state that the use of Corollaries 2 and 4 lead to generalizations which are represented by the original specification.

<u>Theorem 4</u> - Let P be a Type A loop. Suppose f' is the valid generalization of f defined in Corollary 2. Then f' is represented by f.

<u>Proof</u> - Suppose (16) holds and select some arbitrary Y' from $D(f')$ satisfying $B(Y')$. Thus there exists $a1, \ldots, an \in A$, $n >= 0$ and $Y \in D(f)$ such that

$$Y' = a1\$(a2\$(\ldots \$(an\$ Y) \ldots)).$$

By the definition of a Type A loop, we must have $B(Y)$. Applying the definition of f' yields

$$f'(Y') = a1\$(a2\$(\ldots \$(an\$f(Y) \ldots))$$

which is

$$= a1\$(a2\$(\ldots \$(an\$f'(H(Y)) \ldots))$$

by (16) since $B(Y)$ holds. Since $H(Y) \in D(f')$, there exists $b1, \ldots, bm \in A$, $m >= 0$, and $Y1 \in D(f)$ such that

$$H(Y) = b1\$(b2\$(\ldots \$(bm\$ Y1) \ldots)).$$

Furthermore,

$$f'(H(Y)) = b1\$(b2\$(\ldots \$(bm\$f(Y1)) \ldots)).$$

Hence, continuing from above

$$f'(Y') = a1\$(\ldots \$(an\$(b1\$(\ldots \$(bm\$f(Y1)) \ldots))) \ldots)$$

which is

$$=f'(a1\$(\ldots \$(an\$(b1\$(\ldots \$(bm\$\ Y1\ ) \ldots))) \ldots))$$

from the definition of f´. Thus

$$f'(Y')=f'\ (a1\$(\ldots \$(an\$H(Y)) \ldots))$$

which is

$$=f'(H(a1\$(\ldots \$(an\$\ Y\ ) \ldots)))$$

from the uniformity condition (8). Hence

$$f'(Y')=f'(H(Y'))$$

and the theorem is proved.

<u>Theorem</u> <u>5</u> - Let P be a Type B loop. Suppose f´ is the valid generalization of f defined in Corollary 4. Then f´ is represented by f.

<u>Proof</u> - Suppose (16) holds and select some arbitrary Y´ from D(f´) satisfying B(Y´). Thus there exists a1, ..., an ∈ A, n>=0 and Y ∈ D(f) such that

$$Y'=\ a1\$(a2\$(\ldots \$(an\ \$\ \ Y\ \ ) \ldots)).$$

We make the assumption that B(Y). Otherwise, by the definition of a Type B loop, the term an $ Y can be replaced by another Y ∈ D(f). Since B(Y´), this process can be continued until Y´ is written in the form above, with Y ∈ D(f) and B(Y). Applying the definition of f´ yields

$$f'(Y')=f(a1\$f(a2\$f(\ldots \$f(an\$f\ (\ Y\ )) \ldots)))$$

which is

$$=f(a1\$f(a2\$f(\ldots \$f(an\$f'(H(Y))) \ldots)))$$

by (16) since B(Y) holds. Since H(Y) ∈ D(f´), there exists b1, ..., bm ∈ A, m>=0, and Y1 ∈ D(f) such that

```
    H(Y) =bl$(b2$( ... $(bm$Yl) ... )).
```

Furthermore,

```
  f'(H(Y))=f(bl$f(b2$f( ... $f(bm$f(Yl)) ... ))).
```

Hence, continuing from above

```
 f'(Y')=f(al$f( ... $f(an$f(bl$f( ... $f(bm$f(Yl)) ... ))) ... ))
```

which is

```
       =f'  (al$( ... $(an$(bl$( ... $(bm$Yl) ... ))) ... ))
```

from the definition of f'. Thus

```
 f'(Y')=f'  (al$( ... $(an$H(Y)) ... ))
```

which is

```
       =f'(H(al$( ... $(an$ Y  ) ... )))
```

from the uniformity condition (8). Hence

```
 f'(Y')=f'(H(Y'))
```

and the theorem is proved.

<u>Example</u> <u>7</u> - Consider the exponentiation program of Example 4. The generalization obtained from Corollary 2 is

```
       f'(w,c,d>=<w*(c^d),c',d'>.
```

Since f' is represented by f, the iteration condition corresponding to (16)

```
  d>0 & odd (d) -> c^d = c*((c*c)^(d/2))  &
  d>0 & even(d) -> c^d = (c*c)^(d/2)
```

can be used in place of that corresponding to (17)

```
  d>0 & odd (d) -> w*(c^d) = (w*c)*((c*c)^(d/2))  &
  d>0 & even(d) -> w*(c^d) = w*((c*c)^(d/2)).
```

The benefits of this simplification are more striking for more complex types of key variables. To illustrate, consider the program to compute Ackermann's function in Example 6. The generalization obtained from Corollary 4 is

$$f'(\langle s(n),s(n-1), \ldots ,s(1)\rangle) =$$
$$\langle A(s(n),A(s(n-1), \ldots ,A(s(3),A(s(2),s(1))) \ldots))\rangle.$$

Since f' is represented by f, the iteration condition

$$m=0 \rightarrow \langle A(m,n)\rangle=\langle n+1\rangle \quad \&$$
$$m\neq 0 \ \&\ n=0 \rightarrow \langle A(m,n)\rangle=\langle A(m-1,1)\rangle \quad \&$$
$$m\neq 0 \ \&\ n\neq 0 \rightarrow \langle A(m,n)\rangle=\langle A(m-1,A(m,n-1))\rangle$$

can be used in place of

$$s(2)=0 \rightarrow$$
$$\langle A(s(n),A(s(n-1), \ldots ,A(s(3),A(s(2),s(1))) \ldots))\rangle=$$
$$\langle A(s(n),A(s(n-1), \ldots ,A(s(3),s(1)+1) \ldots))\rangle \quad \&$$
$$s(2)\neq 0 \ \&\ s(1)=0 \rightarrow$$
$$\langle A(s(n),A(s(n-1), \ldots ,A(s(3),A(s(2),s(1))) \ldots))\rangle=$$
$$\langle A(s(n),A(s(n-1), \ldots ,A(s(3),A(s(2)-1,1)) \ldots))\rangle \quad \&$$
$$s(2)\neq 0 \ \&\ s(1)\neq 0 \rightarrow$$
$$\langle A(s(n),A(s(n-1), \ldots ,A(s(3),A(s(2),s(1))) \ldots))\rangle=$$
$$\langle A(s(n),A(s(n-1), \ldots ,A(s(3),A(s(2)-1,A(s(2),s(1)-1))) \ldots))\rangle.$$

5. Recognizing Uniformly Implemented Loops

Although the problem of recognizing uniformly implemented loops is in general an unsolvable problem, the following guidelines seem useful in a large number of situations.

Recognizing uniformly implemented loops can be viewed as a search for an operation wrt which the loop is uniformly implemented. In practice, condition (3) is the most demanding constraint on this operation. An effective strategy, therefore, is to use (5) as a guideline to suggest candidate operations. Conditions (6) and (7) must be proven to show the loop is uniformly implemented wrt some particular candidate.

Often the modification to the key variable z in the loop body is performed by a statement of the form

$$z := z \# g(X)$$

for some dyadic operation # and function g. In this case, condition (5) suggests the loop may be uniformly implemented wrt # or some directly related operation. For example, if # is associative, condition (5) holds for #. If # satisfies

$$(a \# b) \# c = (a \# c) \# b$$

(e.g. subtraction), and an inverse #´ of # exists satisfying

$$a \# b = c <-> b \#' c = a$$

(e.g. addition if # is subtraction), condition (5) holds for #´.

Another commonly occurring case is when the future values of the key variable z are independent of X, i.e.

$$h'(z,X1) = h'(z,X2)$$

for all z, X1 and X2. This situation arises most frequently when z is some data structure which varies dynamically as the loop iterates. Typically, there exists some particular aspect or portion of the data structure (e.g. the top of a stack, the end of a sequence, the leaf nodes in a tree) which guides its modification. A useful heuristic which can be employed in this circumstance is to consider only operations which maintain (i.e. keep invariant) this particular aspect of the data structure.

Selecting such an operation $\$'$ guarantees that the "change" experienced by the data structure in the loop body will be independent of any modification $\$'$ and thus insures condition (5) holds.

In any case, recognizing uniformly implemented loops and determining the operation wrt which they are uniformly implemented is often facilitated if the intended effect of the loop body (as regards the key variable) is documented in the program source text. Such documentation abstracts what the loop body does from the method employed to achieve this result and thus makes analysis of the loop as a whole easier.

To illustrate, consider the following program to compute the maximum value in a subarray a[i..n] of natural numbers:

```
m := 0;
while i <= n do
   if m < a[i] then m := a[i] fi;
   i := i + 1
   od
{m = MAXIMUM(a,i0,n)}.
```

If the effect on m in the loop body were documented as

m := MAX(m,a[i]),

its updating would be of the form m := m # a[i] and the heuristic discussed above could be employed to help determine that the loop is uniformly implemented wrt # = MAX.

7. Related Work

The first work on generalizing functional specifications for loops appears in [Basu & Misra 76]. These results are refined in

[Misra 78] and are studied in considerable detail in [Misra 79]. The major contribution of this research seems to be the identification of two loop classes or schemas which are "naturally provable." The first class is called the accumulating loop schema and can be viewed as a (commonly occurring) special case of the Type A loops discussed here. Specifically, a program in the accumulating loop schema with associative binary operation $\$'$ in the sense of [Basu & Misra 76] is necessarily uniformly implemented wrt $\$'$ and meets the criterion for a Type A loop presented here.

The second of these classes is called the structured data schema. A loop in this class is uniformly implemented wrt an operator which adds an element to the data structure being processed in such a way that it is not the "next" element to be removed from the structure (e.g. recall the use of ADDUNDER in the tree traversal example). A loop in this class necessarily meets the criterion for a Type B loop presented here. The program to compute Ackermann's function does not fit in the structured data schema. We remark that the analysis presented here relies on the loop body computing a function, i.e. it relies on the loop body being deterministic. Consequently, the above comments do not apply to the non-deterministic structured data loops analyzed in [Misra 79].

In [Misra 79] the author states that the important common feature between these program classes is that " ... they act upon data in a 'uniform' manner; changes in the input data lead to certain predictable changes in the result obtained." The work we have described can be viewed as an attempt to characterize this commonality and to generalize the work in [Misra 79] based on this characterization.

More recently, [Basu 80] considers the problem of generalizing loop specifications and uses the idea of a loop being "uniform over a linear data domain." One difference between this work and that presented here is that Basu considers only programs in the accumulating loop schema (in the sense of [Basu & Misra 76] without the closure requirement). More importantly, Basu's idea of uniform behavior is based on the behavior of the loop as a whole and seems to be largely independent of the loop body. Our approach relies solely on the characteristics of the loop body.

Misra points out in [Misra 78, Misra 79] that the iteration condition for his structured data schema can be simplified in a manner similar to that presented here; our results show that the same simplification can be applied to his accumulating loop schema. Again, an appropriate view of our research is one of generalizing this earlier work by investigating the theory which underlies these phenomenon.

3. Summary and Conclusions

It is felt that the key to reading, understanding and verifying program loops is generalizing the behavior of the loop over a restricted set of inputs to that over a more general set of inputs. The view of this generalization process presented here is one of ascertaining how changes in values of particular input variables affect the subsequent computation of the loop. This process is facilitated if these changes correspond to particularly simple modifications in the result produced by the loop body.

Of course, the simplest possible modifications in the result produced by the loop body would be no modifications at all, i.e. the output of the loop body (and hence the loop) is completely independent of changes in these input variables. This situation, however, occurs rarely in practice since it implies that the input values of these variables serve no purpose in view of the intended effect of the loop. It is felt that the definition of a uniformly implemented loop presented here is the "next best" alternative, and yet a large number of commonly occurring loops seem to possess this property. The definition states that in terms of the execution of the loop body, prescribed changes in the input value of the key variable affect only the final value of the key variable; all other final values are independent of the change. Just as importantly, the modification caused in the final value of the key variable is necessarily the same as the change in its corresponding input value. This property is analogous to that possessed by a function of 1 variable with unit slope in analytic geometry: increasing the input argument by some constant causes the function value to be increased by exactly the same quantity. Taken together, these factors account for the pleasing symmetry between S and H in condition (8).

Generalizing Specifications For Uniformly Implemented Loops

Viewed as a verification technique for uniformly implemented loops, the procedure described here can be thought of as transforming the problem of discovering the general loop specification into the problem of discovering the operation with respect to which the loop is uniformly implemented. Clearly, this is of no benefit if the latter is no easier to solve than the former. In many cases, however, it seems that simple syntactic checks are sufficient for identifying this operation. For example, in the tree traversal program, the fact that the loop body does not test the stack for emptiness [Basu & Misra 76] is a sufficient condition for the loop being uniformly implemented with respect to ADDUNDER.

It is felt that the notion of uniformly implemented loops may have an application in the program development process. Specifically, when designing an initialized loop to compute some function, the programmer should attempt to construct the loop in such a way that it is uniformly implemented with respect to some easily stated operation. Our work indicates that these loops are susceptible to a rather routine form of analysis. Furthermore, implementing a loop in a uniform fashion requires maintaining a certain amount of independence between program variables (or perhaps portions of program variables in the case of structures) and a simple dependence between the input/output values computed by the loop body. Such programs are desirable since the ease with which a loop can be understood depends largely on the complexity of the interactions and interconnections among program variables. We remark that the question of whether a given program is "well structured" has been viewed largely as a <u>syntactic</u> issue (e.g. use of a restricted set of control structures); we offer the definition of a uniformly implemented loop as an attempt at a characterization of a <u>semantically</u> well structured program.

9. References

[Basu 80]
Basu, S. A Note on Synthesis of Inductive Assertions, IEEE Transactions on Software Engineering, SE-6 (January, 1980).

[Basu & Misra 75]
Basu, S. and Misra, J. Proving Loop Programs, IEEE Transactions on Software Engineering, SE-1 (March, 1975).

[Basu & Misra 76]
Basu, S. K. and Misra, J. Some Classes of Naturally Provable Programs, Proc. 2nd International Conf. on Software Engg., San Francisco, Oct. 1976.

[Gries 79]
Gries, D. Is Sometime Ever Better Than Alway?, Transactions on Programming Languages and Systems, Vol. 1, No. 2, Oct 1979.

[Hoare 69]
Hoare, C. A. R. An Axiomatic Basis for Computer Programming, CACM, 12 (October 1969), pp. 576-583.

[Mills 72]
Mills, H. D. Mathematical Foundations for Structured Programming, IBM Federal Systems Division, FSC 72-6012 (1972).

[Mills 75]
Mills, H. D. The New Math of Computer Programming, CACM, 18 (January 1975).

[Misra 78]
Misra, J. Some Aspects of the Verification of Loop Computations, IEEE Transactions on Software Engineering, SE-4 (November 1978), pp. 478-486.

[Misra 79]
Misra, J. Systematic Verification of Simple Loops, University of Texas Technical Report TR-97, March 1979.

[Morris & Wegbreit 77]
Morris, J. H. and Wegbreit, B. Subgoal Induction, CACM 20 (April 1977), pp. 209-222.

[Wegbreit 77]
Wegbreit, B. Complexity of Synthesizing Inductive Assertions, JACM, Vol. 24 (July 1977), pp. 504-512.

END
DATE
FILMED
1-82
DTIC